N92-24430

TECHNIQUE FOR USING A GEOMETRY AND VISUALIZATION SYSTEM TO MONITOR AND MANIPULATE INFORMATION IN OTHER CODES

Thomas P. Dickens
Aerodynamics Configuration Computing
Boeing Computer Services
Seattle, WA

ABSTRACT

A technique has been developed to allow the Boeing Aero Grid and Paneling System (AGPS[1,2]), a Geometry and Visualization System, to be used as a dynamic real-time geometry monitor, manipulator, and interrogator for other codes. This technique involves the direct connection of AGPS with one or more external codes through the use of Unix pipes. AGPS has several commands that control communication with the external program. The external program uses several special subroutines that allow simple, direct communication with AGPS. The external program creates AGPS command lines and transmits the lines over the pipes or communicates on a subroutine level. AGPS executes the commands, displays graphics/geometry information, and transmits the required solutions back to the external program. The basic ideas discussed in this paper could easily be implemented in other graphics/geometry systems currently in use or under development.

INTRODUCTION

Many computational fluid dynamics (CFD) codes use iterative techniques, or at least a complex solution process that users would like to monitor to allow them to see convergence histories, perturbations to geometry, or other algorithmic processes. Considerable effort would be required to write graphical monitoring code for each such application. Also, many CFD analysis codes have geometry operations written inside the codes. This is frequently done without using the latest state-of-the-art in curve and surface mathematics and without the benefit of the trial and error of investigating variations on a method. These types of problems can be solved quickly with features provided within AGPS.

Programs monitored graphically with AGPS may provide early and detailed insight allowing refinement of algorithms in both precision and in performance. This will be illustrated by the use of AGPS connected to a supersonic marching code running in an inverse design mode.

Beyond passively monitoring codes, the technique allows AGPS to interact with codes, doing geometry tasks that have been perfected within AGPS. This allows rapid prototyping of advanced geometry manipulation and interrogation in other codes without the time-consuming process of building these capabilities from scratch.

OVERVIEW OF AGPS

While developed and used primarily for the preliminary design of aircraft, AGPS is used throughout The Boeing Company for a variety of tasks. The program is a surface geometry system in which virtually any shape can be modeled. The underlying mathematics include cubic and quintic polynomials and rational b-splines for representing curves, surfaces, and solids. The interface consists of a structured programming language with over 150 geometry-related commands, along with a mouse-menu driven interface. Higher-level command files can be constructed and used as a macro capability to do complex or repetitive tasks very simply. Collections of command files are used as high-level packages to allow users to accomplish complex tasks by following an interactive menu-driven session. Hundreds of existing command files are included in the AGPS release, which runs on a variety of machines including VAX, SGI, HP/Apollo, IBM, and Cray. AGPS has been coded to be machine/architecture independent and utilizes a dynamically allocated object data structure. For further information on AGPS, refer to A. E. Gentry's paper, "Requirements for a Geometry Programming Language for CFD Applications",[3] included in these proceedings.

CONCEPTUAL DESIGN OF NEW TECHNIQUE

A short time ago, techniques were developed within AGPS to have a small parent process launch AGPS using a pair of named pipes for communication. When AGPS needs to spawn a new process, the small launch process is signaled to do the task. This is necessary to get around the UNIX problem of using fork and exec, where fork will duplicate the current process' entire program space, then replace it with a new process. With a large number of objects in the data structure, duplicating AGPS would occasionally fail in the fork call. This technique inspired me to develop a new AGPS command, Connect to External Process (CXP).

The CXP command allows AGPS to be driven from an external process with all user input and output routed from and to the external process (Figure 1). The communication is routed through the AGPS launch process, since the communication protocol has already been established there. The simplest form of an external process is a terminal window which passes all strings typed into it to AGPS and echoes all strings which AGPS returns. From here, a user program can construct the individual AGPS commands to accomplish a requested task, providing an on-the-fly dynamic command file generation capability. In addition to command string communication, the external process has access to AGPS internals via (currently) 80+ "subroutine calls", covering areas of AGPS graphics, data structures, memory management, and objects. This is intended to provide enough hooks into AGPS to allow a large variety of tasks to be accomplished easily from within a host of user applications.

The CXP command handles the setup, initialization, and status of communication of an external user program to AGPS. CXP will do as little or as much as needed, from connecting to, or disconnecting from, existing pipes, to creating the required pipes and spawning the specified external process. Once AGPS has connected to the external process and it is active, all input and output is received from and routed to the external process. In other words, the external process takes control of AGPS user I/O.

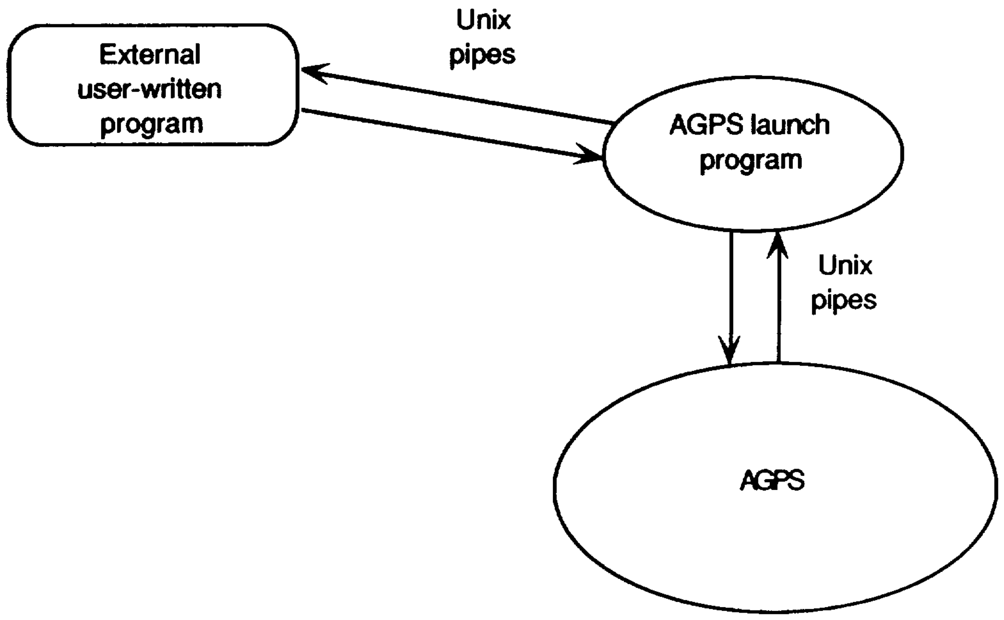

Figure 1. Connection of External Process to AGPS.

TECHNIQUES

Following suit with the techniques used for the AGPS launch process, the connection of AGPS to the external user process is done using Unix named pipes. The pipes are used according to methods outlined in many texts, such as "Unix System Programming",[4] and provides the communication path. The protocol for communication through the pipes was established using fixed-length packets containing command, mode, and data information. Using status commands, a handshaking dialog between AGPS and the external process can be achieved. This quickly became quite involved as more capabilities were explored. While I could get this newly developed subsystem to perform, I soon realized that AGPS users would not want to work at this level of detail. Therefore, I wrote higher-level support routines to do all of the low-level protocol work, allowing a user to simply add a few subroutine calls to their program. These calls took the following form:

connect_2_agps: Create two named pipes, then launch AGPS, instructing it to connect to the pipes and wait for instructions.
command_2_agps: Send the specified character string to AGPS as an AGPS command.
wait_4_agps: After sending a command to AGPS, wait for AGPS to respond. Resulting output from AGPS is also handled.
disconnect_agps: Terminate the AGPS session, disconnect and delete the pipes.

With these four subroutines, usable from either FORTRAN or C, programmers can connect their programs to AGPS and use it to do tasks for their own programs. This follows the AGPS philosophy of providing capabilities to users without restricting or needlessly focusing their use. We have found that users are very imaginative, and many of the currently advertised capabilities using AGPS have been user derived.

When using this technique, it is important to remember that even though your program and AGPS are talking, they are two separate processes and do not share any common memory. If your program has a geometric entity defined, that definition needs to be transmitted to AGPS before AGPS can operate on it, and any results from AGPS need to be transmitted back. To accomplish this, some of the file I/O commands in AGPS were modified to write through the pipes in addition to writing to files.

APPLICATION OF THE METHODS

One application of these methods is to have a user's program transmit some data in the form of geometric objects to AGPS and have AGPS display the objects graphically. This is useful to many scientific programmers who are experts at CFD work or other complex fields, but who don't have the time or the inclination to learn to program graphics on their systems. Using the four above-mentioned pipe commands, they can quickly generate graphics from their programs. During the alpha testing of the CXP capabilities, I used these methods to prototype new view rotation routines, which have been incorporated into the production version of AGPS. I then looked for an interesting program to further validate the method. That case study is highlighted here and represents a part time effort over a couple of weeks.

The program is a supersonic marching code being used for an inverse wing design. Since the flow is supersonic, the effects to changes in the geometry are only felt downstream, allowing the program to simply march downstream. The code was using NPSOL, a general-purpose, constrained, non-linear optimization code based on Stanford University's SOL/LSSOL, SOL/NPSOL, and SOL/QPSOL libraries. At each marching station we used NPSOL to perturb a small number of points in a cross-section of the wing, while attempting to minimize the difference from the current calculated pressure and a specified target pressure. We observed the results of NPSOL to be marginal and very slow (probably, the way we used NPSOL could explain some of this), so we decided to watch NPSOL at work. The marching code was modified to use the four subroutines mentioned above, and a special subroutine was written to construct AGPS commands to represent the desired data in an AGPS geometry format. The data, a composite showing the desired pressure, the current pressure, and the geometry deltas that were applied to the wing section, was displayed and updated in real time by AGPS as the marching code was running (Figure 2).

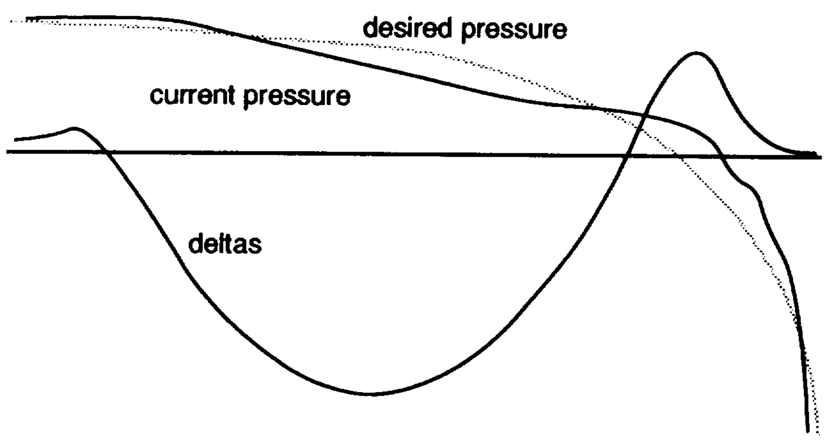

Figure 2. NPSOL Design Status at an Early Iteration.

Each iteration of NPSOL changed the geometry at one design point, then called the user-supplied routine to adjust the grid and recompute the pressure. The resulting plots were very interesting. The current pressure took wild swings from iteration to iteration (Figure 3). After 40 to 50 iterations, things stabilized, and after 120 iterations the objective function was satisfied and NPSOL finished (Figure 4). The slowness was obviously due to the high number of iterations involved, since a refitting of the grid and a pressure solution were needed for each of the 120 iterations for each design station. A higher number of independent design points would be required to achieve a better, smoother convergence to the desired pressure. This would have greatly increased the number of iterations required for each design station, causing the convergence time to be unacceptable.

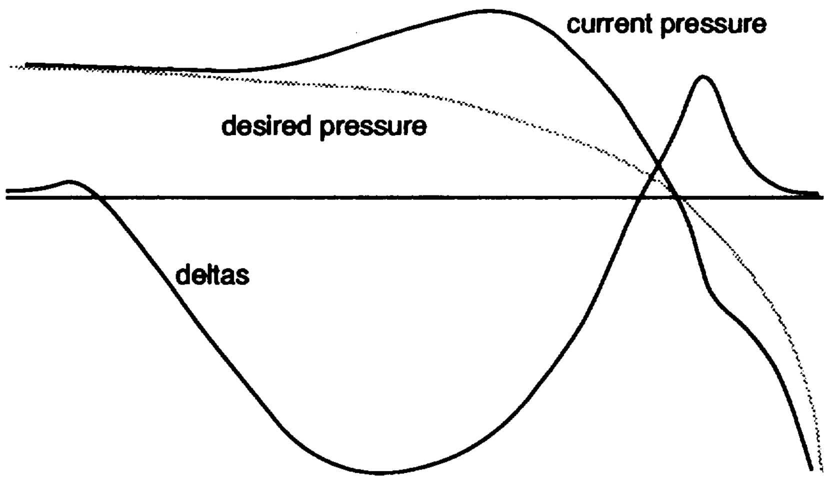

Figure 3. NPSOL Design Status at Iteration 7.

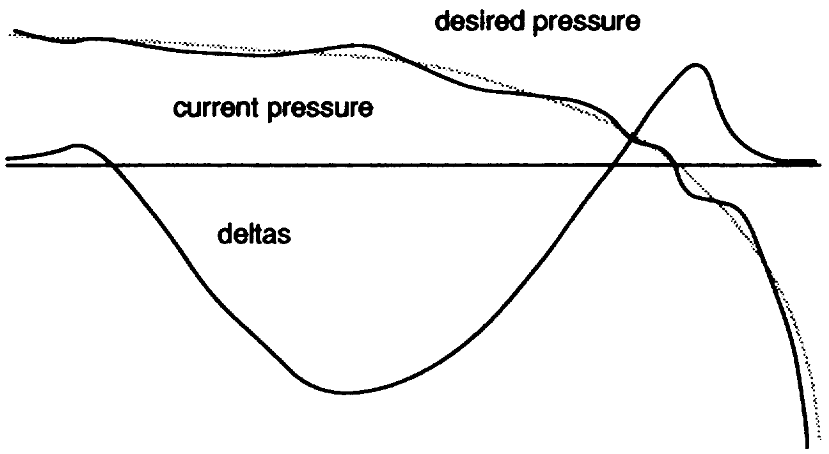

Figure 4. NPSOL Design Status When Completed.

The resulting pressure from the new geometry was similar to the desired pressure, but we observed many wobbles and kinks. When this was shown to an aerodynamicist on the project, he sketched in a number of vertical lines where the current pressure crossed the desired pressure. Remembering his elementary aerodynamics, he then sketched arrows showing the needed movement in the section of geometry to better achieve the desired pressure (Figure 5). Knowing that the flow was supersonic, all of the surface grid points could be intelligently perturbed at the same time, then a new grid could be fit and a pressure solution calculated for the station. Being a computer scientist and not an aerodynamicist, I trusted his observation on blind faith and concluded that I could easily write a subroutine to implement his assertion. I began coding a routine, initially ignoring the fact that the pressure computations occurred on cell centers, while I was applying calculated deltas to grid points. I also ignored the effects of the direction of the velocity vector at the cell midpoints. After trial and error adjustments on the function to calculate the weight deltas, I was able to achieve the desired pressure to within a tolerance much greater than the NPSOL method with fewer than six iterations. The resulting pressure was so close to the desired pressure that a plot of it here would show a single pressure curve.*

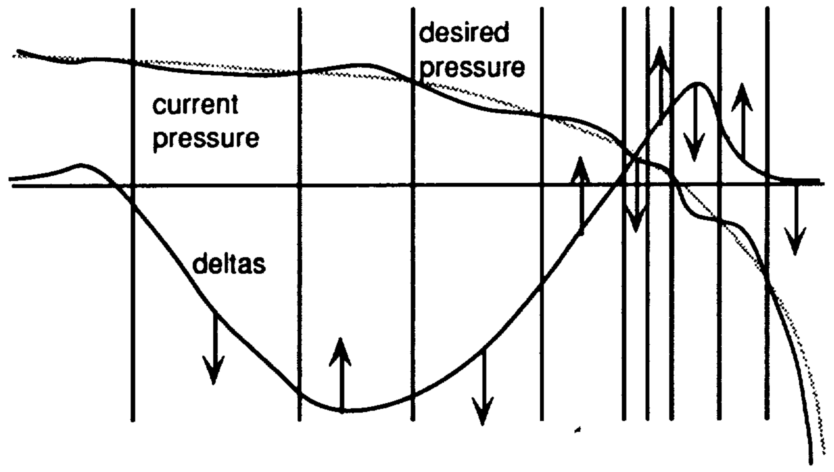

Figure 5. Modifications to the Geometry Deltas to Achieve the Desired Pressure.

We were pleased that in addition to demonstrating a successful application of the AGPS CXP command, we improved the supersonic marching code by more than an order of magnitude in both performance and quality. We were able to use AGPS as a microscope and dynamically dissect and improve this marching code. Now, what else could we do with this?

While we had reached the desired pressure in the design region, there was a problem fitting the new geometry into the old. We experimented in the marching code using crude feathering techniques, but wanted to use more sophisticated methods. We then thought of shipping the "before" and "after" geometry to AGPS, to use proven AGPS methods to blend and smooth the geometry, then ship the results back to the marching code. This was implemented with the smoothing logic residing in an AGPS command file, which could be edited live as the marching code was running. The changes would then be used and seen during the next iteration. The strengths of both codes, the graphics and geometry knowledge of AGPS and the flow-solving knowledge of the marching code, enabled rapid development of the algorithm.

* A short time after these experiments, a coworker in the lab showed us a 1991 paper by W. H. Mason^5 detailing this inverse method we had just implemented. Mason explores much more detail and theory than the implementation presented here.

LIMITATIONS AND FINDINGS

While experimenting with the design code, I timed various operations and tried various things to tune the system. Initially I would construct the geometry inside of AGPS by sending literally hundreds of commands through the pipe to define the desired points, strings, and curves, then by sending the commands to draw the objects. However, the bandwidth of the pipe communication created a bottleneck. I achieved better performance by writing the geometry commands into a file, then sending AGPS the command to read and execute the newly created command file. I admit that I prefer the idea of sending all the data through the pipes, but considering performance, intermediate files are a good solution.

I also investigated using named pipes with an NFS mount to communicate between two machines. After unsuccessful attempts and additional research,[6] I found that even though the named pipes exist on the disk, which allow them to be opened through the NFS mount, the actual communication occurs within the Unix kernel. The packets of data are routed within the kernel from process to process and cannot be passed to another machine.

I then looked into using sockets. The creation of sockets is different than pipes, but they are read from and written to in the same manner as pipes, allowing the same code to do the communication. With sockets however, I experienced performance degradation of two orders of magnitude. Therefore, I recommend using sockets only when the amount of data is small and the benefit of running on two machines is needed. I also experienced occasional sporadic results with sockets involving incomplete packets of data.

CONCLUSIONS

I have described a technique for graphically monitoring codes and performing geometric functions. The technique provides insight to the workings of the codes, and facilitates the task of developing algorithms for them. Additional development time is saved by having AGPS perform geometric tasks, rather than developing and coding these functions independently.

FUTURE WORK

I am interested in refining the use of sockets to offer distributed setups. The developed protocol allows multiple AGPS sessions to be used from a single user's code, which, when combined with sockets, would allow a central program to utilize multiple geometry tasks across a variety of machines. Enhancements to the protocol have been proposed to use an external program as a subsystem of AGPS rather than to control AGPS. Multiple external processes will be able to be dynamically connected to AGPS and accessed as AGPS commands. Subroutine access through the pipes will allow direct manipulation of the AGPS data structure and memory. This can also be used to prototype future AGPS commands and capabilities. As for the future of the inverse design code, ongoing work is being done, with AGPS now an integral part of the effort.

REFERENCES

1. D. K. Snepp and R. C. Pomeroy, "A Geometry System for Aerodynamic Design," AIAA/AHS/ASEE Aircraft Design, Systems, and Operations Meeting, AIAA-87-2902, September 14-16, 1987.

2. W. K. Capron and K. L Smit, "Advanced Aerodynamic Application of an Interactive Geometry and Visualization System," 29th Aerospace Sciences Meeting, AIAA-91-0800, January 7-10, 1991.

3. A. E. Gentry, "Requirements for a Geometry Programming Language for CFD Applications", NASA Workshop on Software Systems for Surface Modeling and Grid Generation, NASA CP- 3143, April 1992.

4. Keith Haviland and Ben Salama, "Unix System Programming", Addison-Wesley, 1987.

5. W. H. Mason, "A Three-Dimensional Inverse Method for Supersonic and Hypersonic Body Design", AIAA 9th Applied Aerodynamics Conference Volume 2, AIAA-91-3325-CP, September 23-25, 1991.

6. M. J. Bach, "The Design of the Unix Operating System", Prentice-Hall Software Series, 1986.